Project Integration Architecture: Inter-Application Propagation of Information

Dr. William Henry Jones
National Aeronautics and Space Administration
John H. Glenn Research Center at Lewis Field
Cleveland, OH 44135
216-433-5862
William.H.Jones@grc.nasa.gov

X00.00 15 Nov 1999
X00.01 20 Sep 2000

Keywords:
PIA; PRICE; CORBA; Knowledge Management; Application Integration; Technical Integration Technologies; Semantic Encapsulation; Self-Revelation; Propagation of Information; Application Graphs;

ABSTRACT: *A principal goal of the Project Integration Architecture (PIA) is to facilitate the meaningful inter-application transfer of application-value-added information. Such exchanging applications may be largely unrelated to each other except through their applicability to an overall project; however, the PIA effort recognizes as fundamental the need to make such applications cooperate despite wide disparaties either in the fidelity of the analyses carried out, or even the disciplines of the analysis. This paper discusses the approach and techniques applied and anticipated by the PIA project in treating this need.*

1 Introduction

Any significant engineering project is inevitably the subject of multiple analyses of various parts of the overall whole. For example, the design of a new air vehicle must consider all manner of things from the external aerodynamics experienced in the extremes of its flight envelope to the heat absorbtion characteristics and combustion performance of its fuel. Seldom are all the engineering concerns of a project captured in a single analysis. Indeed, the normal experience is that each analysis is narrowly focused to a single discipline and to a sometimes-ill-defined level of fidelity.

Increasingly, these analyses are captured as computer programs of one form or another. Predictably, the programs display a nearly limitless variety of forms for their input, output, and operational expectations, as well as any other characteristic that might be meaningfully defined. A foundational goal of the Project Integration Architecture (PIA) was to provide a single, common, object-oriented wrapping layer that could buffer this endless diversity of formulation and make all things appear the same in some useful way. It is felt that this goal has been reasonably achieved and demonstrated in a C++ language implementation. The architectural result is discussed in a previous paper [1].

The PIA project recognized, though, that such disparate analyses were seldom applied to an engineering project in isolation. For example, an extremely effective fuel in terms of combustion performance probably affects the structural analysis of the vehicle in that its containing tank structure may be smaller and lighter, or an extremely low density fuel may affect the external aerodynamic analysis by requiring the accommodation of greater bulk.

Traditionally, such analytical couplings have been handled in a manual way, often simply through human-to-human interactions, and have resulted in overall engineering analyses that were less sophisticated than might have been desired. To contribute in this area, the PIA effort has always intended to build upon the common foundational-architecture base to facilitate the automated flow of meaningful information between such applications.

A key element in the expectation that this goal could be achieved was the perception that disparate analyses could meaningfully transmit information content at some basic, physical level. For example, the expectation was that two computational fluid mechanics codes, despite very different formulations of the flow field problem, would, nevertheless, be able to communicate meaningfully through basic statements supplying things such as total temperature, total pressure, velocity vector, and the like, all given throughout some interfacing volume. That expectation comes from the realization that, were two engineers sheparding these disparate, yet cooperating, analyses for a project, such is the nature of the communications that would be had. Thus, a part of the PIA contribution is to provide a place where such basic transfers can be encoded and automated.

Another aspect of traditional coordination of cooperating engineering analyses is that, at some small but persistent statistical rate, the product configuration being analysed gets out of synchronization. The aeroperformance of the thin wing is analysed and combined with the fuel capacity analysis of the thick wing, resulting in a winning, if unmanufacturable, design. The PIA effort solves this coordination problem, too. The solution is, of course, merely a matter of bookkeeping, but such bookkeeping is a matter proven to be best left to the ruthless stupidity of computing machinery.

Finally, it remains to be said that the PIA effort does not solve the propagation of information problem at the semantic level, but only at the generic, bookkeeping level. Semansis is a quality that is void until the architecture is applied to an application. It is only as an application enters the PIA world through the efforts of the person developing the application wrapper that a survey of the information likely to be available may be made and instructions encoded into that wrapper which make reasonable use of those elements of information that, in a semantic manner, make ‘sense’.

2 The Solution

As with many problems in computing, clever (or, for the more modest among us, fortuitous) arrangement of the elements renders the actual solution so simple as to leave the observer questioning why the problem was ever posed. To a considerable degree, this is the situation found in the inter-application propagation of information within the PIA environment. Nevertheless, an effort will be made here to make matters seem complicated and, consequently, impressive.

2.1 The Application Graph

The first challenge to be confronted in the propagation of information between applications is identifying to each such cooperating application just exactly which other applications it might obtain information from and which applications it should distribute its own information to. In treating this issue, the PIA project introduces a condition considered to be life-like, although perhaps not without exception: that composite engineering analyses have a definable flow of information which obeys a concept of causality.

For example, consider a simple case of such information flow in which analysis A provides the information which analysis B needs to go forward, which in turn gives rise to the information for analyses C and D which can go forward independently, each producing information which is finally assessed by analysis E. That final assessment by analysis E may indicate that the proposed design is unsatisfactory and give rise to an adjustment which then starts the analysis flow again.

This causality assumption is a key constraint and allows the PIA project to arrange cooperating applications as a directed, acyclic graph. This is easily done since **PacAppl** application objects inherit the characteristics of directed graph nodes from the **PObjDgn** class layer.

Such a directed graph of cooperating applications is represented (for the present, rather optimistically) by Figure 2.1, a graph of applications which were, themselves, recently used in the analysis of the Rocket Based Combined Cycle propulsion system proposal being developed at the Glenn Research Center. Each labeled box of the figure represents another PIA-wrapped application; CAPRI cross-vendor access to CAD geometry data, NASTRAN structural analysis, trajectory analysis, and so on. In this engineering flow, a selected geometry (as revealed by CAPRI) is the ultimate driver, the initial node of the graph. Geometry then directly feeds to APAS, an aeropanel loads code, GASP, a more refined computational fluids analysis, and NASTRAN, a reliable analysis of structural performance. As the figure shows, information is to flow in expected patterns. Note that the GASP fluid analysis relies not only upon the basic geometry received from CAPRI, but also on the lumped aerofluid analysis of APAS.

The bottom of the figure shows an even more optimistic element: a conflict resolver and system optimizer. Here, the aggregate of engineering analysis is to be brought together, assessed for merit, and, potentially, result in a design alteration represented by the sweeping curve back to the geom-

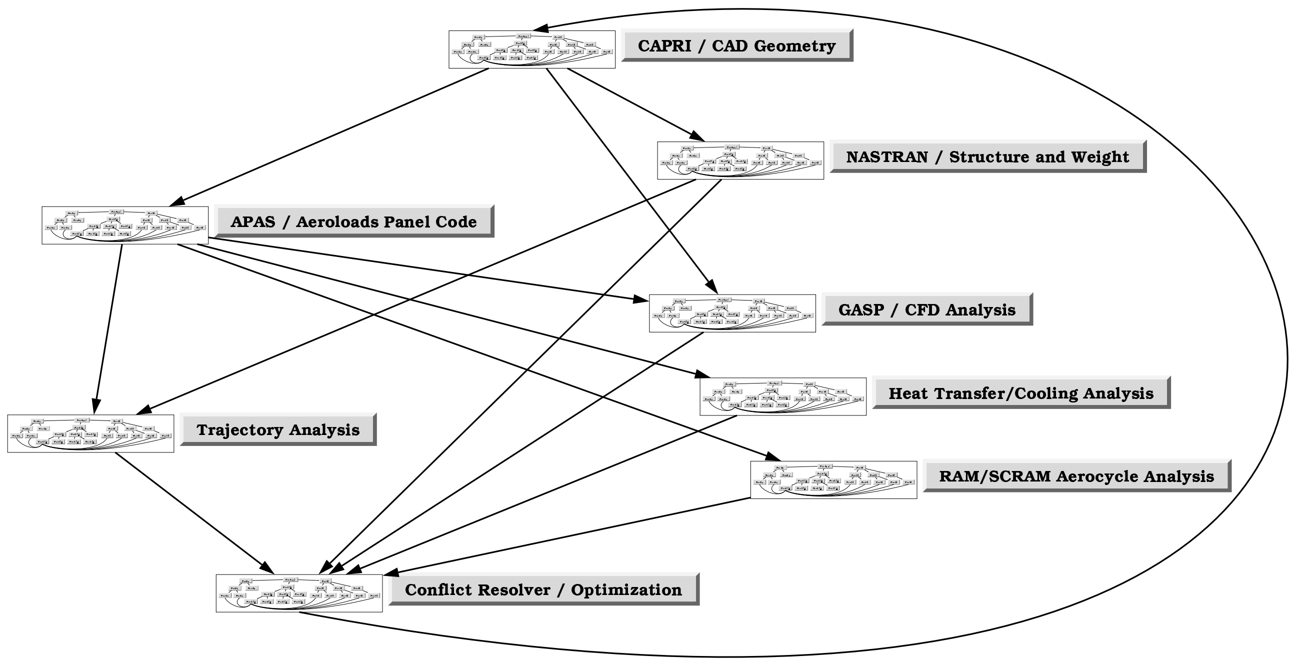
CAPRI / CAD Geometry
NASTRAN / Structure and Weight
APAS / Aeroloads Panel Code
GASP / CFD Analysis
Heat Transfer/Cooling Analysis
Trajectory Analysis
RAM/SCRAM Aerocycle Analysis
Conflict Resolver / Optimization

Figure 2.1: A Graph of Cooperating Applications

etry element at the top of the graph. It must be emphasized that, at the time of the writing of this paper, this final assessment element is only a projection that such a thing can be done. At the moment, this role is filled by the engineer, or more aptly, by the team of engineers effecting the project. Nevertheless, it is important to note that the PIA project has a place for such a thing to go.

It is also important to note that this sweeping curve is not intended to be an actual graph edge making the directed graph cyclic. Instead, it is expected that the resolver/optimizer application will be built with knowledge of the overall problem (often expected to be a problem driven by geometry) and that the feedback operation will be in the manner of a user interaction causing a new configuration of the problem rather than as a cyclic propagation of an existing configuration of the problem.

Information propagation begins in the PIA scheme in response to some (customarily) external event, usually delivered in the context of the initial node of the application graph. That responding application assures that its information state is as complete as possible, and then re-delivers the propagation event to each of its immediate successor applications. Each of those applications, in turn, assures that its own information state is as complete as possible (to be discussed further in a moment) and then re-delivers the propagation event to its own immediate successor applications. This process ripples through the application graph until it reaches all of the terminal nodes of that graph.

(As mentioned above, the application graph is expected to be acyclic. The design revision anticipated in Figure 2.1 by the conflict resolver block and its sweeping curve back to the initial node of the graph is expected to be a special operation of that 'application' rather than a simple information propagation act, even though the propagation code may be cleverly reused.)

Because the application graph is not constrained to be a simple *n*-ary tree (that is, because a particular application, for instance the GASP computational fluids analysis in Figure 2.1, may have more than one primary source of input information), assessing the completeness of an application's information state includes waiting for all of the immediate predecessors of an application to complete their part of the propagation act and transmit the propagation event on to the receiving application.

Also, the PIA implementation allows for the possibility that, while the immediate predecessors of a given application may represent the bulk of the information flow, they cannot be proved to represent the entirety of that flow. Thus, the implementation examines not only the immediate predecessor application transmitting the propagation event, but all the extended predecessor applications of that immediate predecessor application, visiting that graph set in depth-first order for reasons that will be explained a little later. (Here, the extended predecessors of a graph node are considered to be all the nodes reachable by any sequence of backward-flowing predecessor edges originating in the subject node. Also, depth-first order is, simply put, visiting the node most remote (or deepest) from the initial node of a graph first, then the next most remote nodes, etc.)

Finally, the PIA implementation recognizes that, after all possible information sources have been harvested, a computational operation will typically complete an application's information state by producing outputs from the input parameter set. This operation is, of course, application specific; however, the PIA design assigns the operation a place within the application object and always invokes that functionality between the completion of source examination and the further propagation of information to successor applications.

Traversals of the extended predecessor set of the immediate predecessors of a node of a graph will produce redundant examinations of some nodes if multiple immediate predecessors to the node exist. (At a minimum, the initial node will be examined once for each immediate predecessor.) In many situations, redundant examinations will be without useful result and, thus, the implementation provides a facility to prevent them. This facility, though, is optional because there may also exist cases in which particular information may not be assimilated until other coordinated information has been obtained.

2.2 The Data Configuration Graph

Having considered the act of information propagation at the level of the application graph, the next step is to consider the act as it operates within the context of source and destination applications. As discussed in the cited paper [1], PIA applications do not contain a single input/output state vector for a single configuration of the problem at hand, but a multitude of such state vectors for all the problem configurations studied. These state vectors are held by **PacCfg** parameter configuration objects which are, themselves, arranged as *n*-ary trees for a given application. Another key focus of information propagation is to keep these problem configuration trees synchronized so that mismatched analyses do not occur.

Each parameter configuration object is required to have a name unique among its siblings in the parameter configuration tree. This allows the correspondence between two such trees, as in Figure 2.2, to be established. Thus in the figure, it is possible for code to establish that the initial parameter configuration node of the source application and the first two subgraphs emanating from it (proceeding from left to right) correspond to the entire configuration graph of the receiving application. Further, by concatenating the name of any given node with those of its ancestors up to the initial node of the tree, a unique path can be identified which may then be applied to another tree (as far as that tree exists) to identify the corresponding node.

This capability is employed by the implementation in response to a propagation operation (represented somewhat artistically by the sweeping curve of Figure 2.3) when the identified source configuration is not of the receiving application's own parameter configuration tree. Assuming that the cited parameter configuration node of the source application does not already have a corresponding node in the receiver's tree, rigid correspondences between the two trees are established and the attachment point in the receiver's tree is identifed. A node corresponding to the cited node is created and attached, along with nodes corresponding to those of any subgraph the cited node may head. The names of the source parameter configuration nodes are used to name the created, corresponding receiver nodes so that project configuration sychronization is maintained.

Note that the information propagation process will refuse to go forward if the necessary correspondences between parameter configuration trees cannot be established. Propagation processes cannot proceed between applications with disparate parameter configuration trees, at least when those disparaties exist in the path between the initial node and the attachment point. Thus, pre-existing, independent analyses cannot be connected together helter-skelter even though two nodes correspond through their names while their ancestral paths differ. The intent of this is, of course, to establish some modest assurance that the propagation of information from application to application is, in some project configuration sense, reasonable.

Another intention of this propagation not by single nodes of the parameter configuration tree, but by (potentially) subgraphs of that tree is to allow and automate the processing of entire design sets. Such sets might provide for the systematic variation of parameters through technologies such as design of experiments, probablist theories, or even by-guess-and-by-golly speculation. Data configuration tree policies (discussed in the cited paper) provide for the intra-tree replication of subgraphs for just such purposes and the information propagation implementation continues that support throughout the application graph.

2.3 The Data Configuration Node

Setting aside the complexities of propagating entire subgraphs of information, it is now time to consider the mechanisms of propagating information from a single source

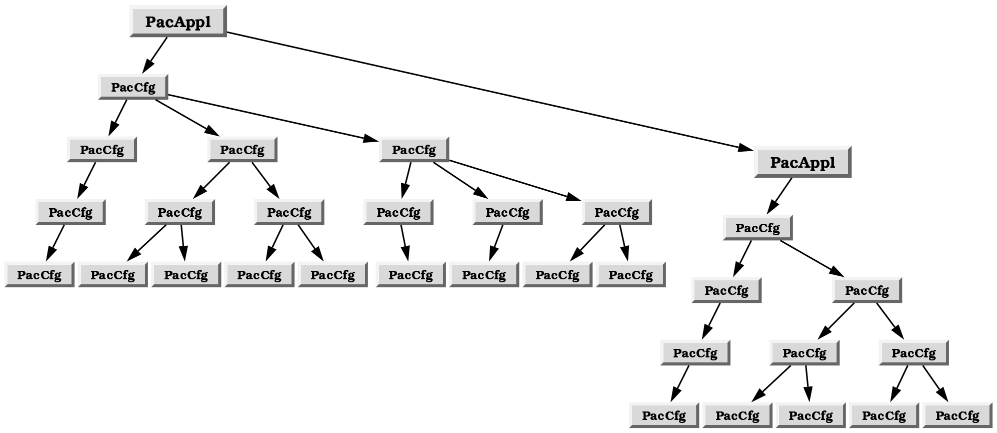

Figure 2.2: Two Applications Prior to Information Propagation

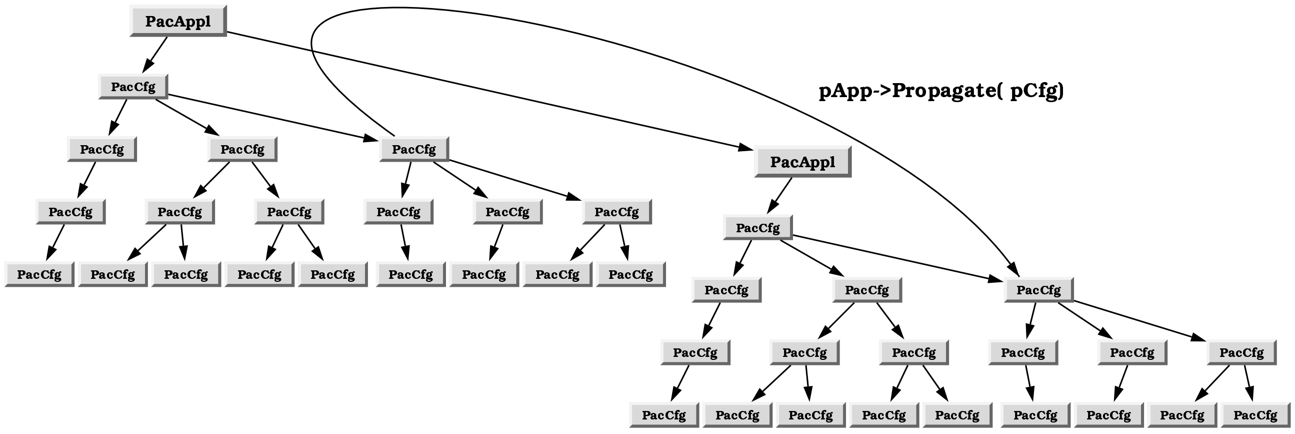

Figure 2.3: Two Application Data Configurations After Information Propagation

parameter configuration node to a single destination parameter configuration node. For each such configuration pair, there are three distinct points or phases at which information may transfer: a preparatory configuration-to-configuration operation, a parameter-by-parameter operation performed for each parameter identified by the source parameter configuration, and a postprocessing configuration-to-configuration operation.

The preparatory and concluding operations are really one philosophically: the opportunity to operate on the whole rather than upon the individual parts. It was supposed that some situations would exist in which the combination of several pieces of information would be required to synthesize a given parameter in the receiving application. It is to meet such a purpose that the preparatory and concluding operations are designed.

Unfortunately, preparatory and concluding operations (as is the parameter-by-parameter operation, too) are entirely semantic in nature and, thus, little constructive beyond declaring their existence can be done; however, one small insight was possible. It was thought foolish to place on these node level operations the burden of a parameter aggregate scan when that was exactly the effort implemented by the parameter-by-parameter operation. To allow the node level operations to capitalize upon the parameter-by-parameter effort, an operating context was defined as existing and persisting through the entire information propagation operation for a given destination node. (Naturally, the base implementation makes this context null, but derived applications are entirely free to place whatever might be needed in this conceptual slot.) The parameter-by-parameter operation may add to this context such parameters as may be appropriate to a particular synthesis. The concluding node level operation may then act upon the parameter harvests that have occurred.

Because this destination-node-level propagation context exists beyond the scope of a particular node-to-node effort, it may harvest parameters from more than one source application. Indeed, the context is free to be a persistent part of the destination parameter configuration and exist through the entire parameter propagation cycle of an application graph, or even beyond that.

The parameter-by-parameter operation is, on its surface, almost self-explanatory. Once a source-destination parameter configuration pair has been established (and the preparatory operation accomplished), a simple iteration through each identified parameter of the source application is performed, offering the destination parameter configuration the opportunity to examine each such parameter and acquire such information as it may from it.

Naturally, things are just slightly more complex than this. Another concept of the parameter configuration tree (detailed, again, in the cited paper [1]) is the inheritance of parameters. In any given parameter configuration, there is the need to insert only the parameters that differ it from its ancestral heritage. A missing but needed parameter causes an ascent of the parameter configuration tree which is stopped by the first configuration that actually has a copy of the needed parameter.

Causing this parameter inheritance mechanism to work in receiving applications after the act of propagation is of some interest in order to preserve the benefits of the concept. To do this, the code that invokes the application specific parameter-by-parameter operation has the ability to avoid the operation if the source parameter to be examined does not, in fact, exist in the source parameter configuration. This avoidance is optional and applications that need to look at every parameter, whether inherited or not (perhaps to form an aggregate for later synthesis), have that option.

2.4 The Parameter Object

With the exception of the application wrapper code, the parameter object is the most situation specific element of the information propagation process. Despite this, two quite general achievements are in the province of the parameter object: the provision of a definitive statement of the semantic meaning it encapsulates and the potential offering of that information in a variety of forms convenient to common usages.

The statement of semantic meaning is not, of course, a programatic entity encapsulated within the parameter object. Instead, the statement is implicit in the object kind. For example, an object that reveals itself through run-time mechanisms to be of the kind ‘parameter, scalar, double, Mach number, far-field’ implicitly reveals its semantic nature to be just that: a far-field Mach number rendered as a scalar double value. This is a small, simple, but nevertheless crucial contribution to the process for it allows consuming propagation code to determine exactly the semantic nature of the information it is looking at.

It is anticipated that a fully-matured PIA object space will have a multitude (indeed, a multitude of multitudes) of defined parameter objects. The propagation code of each application wrapper is to act as a filter upon the variety of parameters it sees; those that make some sense are processed (in some way) while those that bear no relevance are simply ignored as they pass by.

Even this process, though, has its vagaries. A code seeing and consuming, say, a local gas kinematic viscosity located at a position that seems relevant still has no absolute assurance that this local gas kinematic viscosity is the local gas kinematic viscosity that is needed. It is first up to the designer of the application graph to assure (to the extent possible) that encountered information is, in fact, relevant information. Such tricks as may be played with a directed graph may be needed to keep one node from encountering information that is not relevant to it.

Applications may also wish to implement in the propagation coding a sensitivity to information proximity, perhaps presuming that information nearer to it in the application graph is more relevant to it. This is why the depth-first order was selected for the traversal of the application propagation source set. This allows a receiving application to encounter nearer information first, giving the potential that such consumed information can 'blank off' in an inheritance manner more remote information. Consider, for example, an application graph that first applies an Eulerian potential flow solver to a flow field, refines that answer with a Navier-Stokes solver, and then passes that result on to some further consumer. If appropriately programmed, the receiving code can consume the Navier-Stokes solution and, by so doing, 'blank off' the presumably less useful potential flow solution.

All of this runs the risk of making application wrappers that are too situation specific. Continuing the above example, the temptation is to develop an application wrapper that expects always to have a Navier-Stokes solution flowing from a potential flow solution. Such temptations must be resisted and the delicate art of working in response to what comes to hand, rather than in an expectation of what will be found, must be developed and practiced.

Another contribution of parameter objects to the information propagation process may be that some objects may offer their content in a variety of useful forms. Information in parameter objects is to be generic and closely defined in a semantic sense. To be useful to all applications, PIA parameters must not be taylored in their content or organization to the needs of any one particular application. But this does not mean that only one accessible view of that information is permitted to exist. It is entirely reasonable that the parameter object be the place in which alternate modes of access serving diverse needs are encapsulated.

Consider a parameter encapsulating a computational fluid flow grid. In its simplest conceptualization, such a grid is an ordered set of n-tuples specifying computational node locations and state variables in m-dimensional space. As independent work has demonstrated, while this is the essense of such a grid, only a fraction of the computational fluids codes operate with a grid structured in this manner. Some codes insert skip rows, skip columns, or skip planes in their grid organizations. Other codes circularize the set so that element n maps back to element 0. Some codes even change the handedness, operating not in the customary right-handed orientation, but in a left-handed one. As that independent work has shown, all such variations can be well and effectively accommodated by variant access views operating on a single internal formulation.

2.5 Documenting the Flow of Information

The final, small contribution of PIA information propagation is to provide for the documenting of information flow. One of the myriad goals of the project was not only to have all the configurations of a given problem that were considered, but to produce an audit trail that could reveal just exactly where a given configuration came from.

All parameter objects of the PIA environment (and, indeed, very nearly all objects of every sort in that environment) derive from a base class with the characteristic of describability. One of the descriptive forms that may be added to an object's description is a change history, and, with the development of the information propagation capability, one of the elements of such a change history may be an information propagation record.

The information propagation descriptive element is a reasonably simple thing. It provides a traversable set of parameter configuration/identification object pairs, each of which, within the application wrapper architecture, uniquely identify a source parameter used in the synthesis of the described parameter. The provision of a set of such pairs is intended to allow for the operation of source aggregates (discussed above) in which several source parameter objects, possibly harvested from different applications, are combined to produce a single parameter needed by the receiving application.

3 When All Does Not Go Well

The discussion thus far has, of course, dealt in that rose-colored world in which all goes well all of the time. Those with more accurate powers of observation (sometimes called cynics) will point out that this is not always the true nature of engineering and scientific analysis. One might debate the advisability of introducing analyses still in the crash-and-burn phase of program development into the postulated world of automated, multi-fidelity, multi-discipline evaluation. Nevertheless, it is prudent to provide for the exceptional event, even though one hopes that such things will be, well, exceptional.

The PIA project does, in a preliminary way, provide for such operational difficulties, but such discussion has been held to this point in order to reduce the complication of the earlier discussion. There are, in fact, just two key elements: the notation of parameter configurations in which a malfunction of some sort has occurred, and the provision of an event mechanism which transmits to some independent (though in important ways undefined) authority knowledge of the occurrence and solicits from that authority one of several well-defined responses.

3.1 The Origin of Difficulties

The discussion begins with the presupposed originator of such difficulties: the computational operation which converts inputs to outputs for a given application. Within the PIA implementation, this process returns a simple, boolean result: success or failure. Should a failure occur, it is handled in the context of the parameter configuration object within which the computation was performed by invoking a malfunction event facility built into the base class of application objects.

The malfunction event facility works in the hope, if not the expectation, of event mechanism objects being connected to the parameter configuration object. A malfunction event is passed to each such found event object and the response noted. In the event of multiple event objects (giving multiple responses), a simple algorithm merges the responses, in general elevating the composite response to the most dire of the individual responses. In the event that there are no connected event objects, a 'no response' response is defined and is the base value from which the malfunction event mechanism begins.

These event objects, as implemented by the basic architecture, in fact do nothing. They provide form without function. It is left to the operating environment hosting the application graph to provide a derived event object which adds substance to this form. It is in this way that the nature of the independent authority is left undefined. The actual event object may lead to a pop-up dialog in a GUI environment for the user to click at, or it may email somebody at home someplace and wait for a responding message. Nearly any conceivable thing can be laid over this basic skeletal event idea.

3.2 Handling a Decision

Once some response (including the 'no response' response) is obtained in answer to the computational malfunction, three potential handlings are implemented: the malfunction may be ignored, the computation may be retried, or the malfunction may be accepted and operations continued to whatever extent possible. The first two choices are simpler. They either stipulate that, despite appearances to the contrary, everything is, in fact, all right, or that maybe if tried again (perhaps after some corrective action that has occurred in the course of event response), everything will become all right. In either of these cases, the parameter configuration does not receive the malfunctioning characteristic.

In the third choice, that of accepting the unsuccessful operation, the outlying mechanisms of malfunction begin to operate. To begin, the presenting parameter configuration object is given the malfunctioning characteristic. This characteristic is regarded as casting a shadow of doubt upon any parameter of the configuration that has the output characteristic. (Within PIA, parameters may have either an input or output characteristic, or both, or neither.) Such parameters of malfunctioning configurations are often referred to as untrusted.

Within the context of information propagation, the failure of a single computation alone is not considered sufficient grounds for complete abandonment of the overall operation (although this selection is an option). Such a failure may well be an isolated problem of a particular configuration of parameters in a larger propagation process that is progressing meaningfully.

3.3 Dealing with Difficulty

The next challenge is to persevere in the face of such malfunctions. This is begun simply by not breaking the information propagation process. Then, as the process continues, it is usually appropriate to avoid the dependence of further operations upon untrusted parameters, and to note those further application/configuration instances to which such untrusted parameters may have transmitted information.

To this end, the parameter-by-parameter operation screens for parameters with the output characteristic that actually exist in a configuration with the malfunction characteristic. By default, parameters meeting this criteria are not transmitted to the parameter propagation code; however, this screening may be turned off. Without regard to that choice, though, the occurrence in the parameter-by-parameter process of any such untrusted parameter causes the receiving configuration to obtain the malfunctioning characteristic. Thus, a malfunction propagates through the application graph even though receiving applications have no particular fault of their own.

The justification for this aggressive propagation of the malfunction characteristic is this: to be conservatively clear as to just where an untrusted result reaches. To understand the reasoning, consider the alternative: the propagation of the malfunction characteristic might have been limited to just those configurations that actually used an untrusted parameter. Then, by electing to screen out such parameters (which is the default), a receiving configuration would avoid dependence on dubious results and inhibit the propagation of the malfunction characteristic. While this might appear desireable, the difficulty lies in the fact that the configuration would then (probably) inherit corresponding trusted parameters from its ancestral line, or from the ancestral line of its propagation source configurations. This, in turn, would constitute a corruption of the configuration tree requirement since the trusted parameters used for the propagating analysis would not necessarily be those appropriate for the malfunctioning configuration. Thus, it is necessary to make the conservative choice of propagating the malfunction characteristic to any configuration that would have used an untrusted parameter had that parameter not been untrusted.

3.4 Not Making More Troubles

Having propagated the malfunction characteristic on to a receiving parameter configuration, the next step is to understand that it should be customary to avoid the computational conversion of inputs to outputs when that characteristic has, in fact, propagated to the application. The reasoning here is that the act of information propagation is for the purpose of obtaining at least some of the inputs to a receiving application. If those inputs are untrusted (because their sources were untrusted), or if they are missing (even though alternative inputs might be inherited from an ancestral configuration), it is probably unwise to expend the effort of converting erroneous inputs to even more erroneous outputs.

The avoidance of computation need not be an unexcepted rule. The introduction of the semantics of a real application provides the opportunity to qualify the relevance of particular untrusted parameters. The application graph is not a perfect device and may often present parameters, untrusted though they may be, of no relevance whatsoever to a particular receiving application. Thus, a particular application may wish to keep its own accounting of untrusted parameters and make its own assessment of its malfunction state. In the event that the parameters that were significant were, in fact, also good, then it is entirely appropriate for an application to perform the defined computation and reset the malfunction characteristic of the configuration. In this way, a malfunction need not propagate to every application of the graph reachable from the malfunction's point of origin.

4 Summary

This paper has discussed the mechanisms and protocols implemented by the PIA project to effect information propagation between engineering analyses cooperating to form an overall analysis of a given project. The effort at the generic base level is principally one of bookkeeping; particulars must always await the semantics of a specific application. Benefits beyond enabling such composite analyses include the assured synchronization of project configuration among analyses, the ability to synthesize a particular parameter from an source aggregate, and the generation of an auditable trail for the propagated information.

The key contribution of the effort, in the event that it works meaningfully, is that it will allow a wide variety of disparate analyses to be brought together into a sort of super application, regardless of discipline or fidelity. Further than this is the fact that such super applications are easily reconfigurable simply by reformulating instances of their component application wrappers into a different directed application graph.

References

[1] William Henry Jones. Project Integration Architecture: Application Architecture. Draft paper available on central PIA web site, March 1999.